Programming Pearls of Go

How to write fast, beautiful Go code

Aaron Schlesinger
Sr. Architect, Deis
October 1, 2016

About Me

- Go for 3+ years
- C => C++ => Python/PHP/Ruby => Java => Scala/Haskell => Go
- Distributed systems
- Kubernetes (kubernetes.io)
- Go In 5 Minutes (goin5minutes.com)

Context-specific design patterns

Pearls?

What They Are

- Recognizable design patterns
- Guidelines for common tasks in Go
- Practical, not theoretical

What They Are Not

- Boilerplate
- Intended to make your code more verbose
- Stolen from Programming Pearls (highly recommended though!)

Today

- Error Handling
- Time
- Concurrency

Error Types

Build a Package That…

- Returns predictable errors
- Lets the caller introspect those errors
- Helps the caller handle the errors intelligently

Exceptions?

- Go doesn’t have them
- But they’re typed, they include error *context*
- Theoretical benefit, practically can be hazardous

The Error Interface

https://godoc.org/builtin#error

```
type error interface {

    Error() string

}
```

Custom Error Type

```
type errBadNumber int


func (m errBadNumber) Error() string {
    return fmt.Sprintf(“%d was bad”, int(m))
}
```

Let Callers Introspect

```
// IsBadNumberErr returns true when a function encountered a
bad number

func IsBadNumberErr(e error) bool {

    _, ok := e.(errBadNumber)

    return ok

}
```

- Naming scheme: `IsXErr`, where `X = (“ErrX” - “Err”)`
- If error type is un-exported, provide an exported bool func & document its use

Cron for Go

Build a Package That...

- Executes a function every X seconds in the background
- Stops execution after Y seconds have elapsed
- Tells you each time the function executes

Goroutines

Refresher:

- “Cheap” threads
- Dead-simple syntax
- Limited on their own

Time

- https://godoc.org/time
- `time.Ticker` - fire every X seconds
- `time.Timer` - fire after Y seconds

The Interface

```
type tickFn func(int, time.Time)

// execute fn every x until y, sending the iteration number
on ch. Blocks on each send to ch, and closes ch when all
executions complete

func call(x, y time.Duration,

    ch chan<- int, fn tickFn) {

    // TODO: implement

}
```

Usage

```
ch := make(chan int)

go call(1 * time.Second, 10 * time.Second, ch, myFn)

for i := range ch {

    log.Printf(“execution # %d”, i)

}
```

Implementation (I)

Tickers and Timers:

```
timer := time.NewTimer(until)

ticker := time.NewTicker(every)

// both of these use resources like background goroutines.

// call stop to free up those resources

defer timer.Stop()

defer ticker.Stop()
```

Implementation (II)

http://bit.ly/svcc-2016-cron-for-select

Tie It All Together

http://bit.ly/svcc-2016-cron

Ack/Nack

How Do You…

Build a pool of worker goroutines that:

- You can submit work to
- Will tell you when they’re done
- Will (optionally) update you on their progress

Channels

- Saw them in the last example
- Concurrent synchronous (optional) queue
- You can send anything on them, *including other channels*

Channel-Over-Channel

```
workSubmitter := make(chan chan int)

// later on …

recv := make(chan int)

workSubmitter <- recv

result := <-recv // wait until worker is done, fetch result
```

The Worker

```
func wrk(ch <-chan chan int) {
    ret := <-ch
    time.Sleep(time.Second) // simulate work
    ret <- rand.Int()
}
```

Starting a Bunch of Them

Receive on a broadcast channel, return on a point-to-point channel

```
const numWorkers = 100

wrkCh := make(chan chan int)

for i := 0; i < numWorkers; i++ {

    go wrk(wrkCh)

}
```

Putting It All Together

http://bit.ly/svcc-2016-ack-nack

Questions

arschles@gmail.com

goin5minutes.com
deis.com

@arschles
@goin5minutes